1. 上课须知

课程主题：ZooKeeper 第二次课 -- 企业最佳实战
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次课程内容

上一次课的主要内容有：

```
1、课程安排 + 上课约定
2、集中式 分布式 摩尔定律 去IOE 性能指标 一致性级别
3、分布式事务（2PC,3PC）
4、分布式一致性算法（Paxos，Raft，ZAB）
5、鸽巢原理 + NWR读写模型 + Quorum议会制
6、CAP 和 BASE 理论
7、ZooKeeper 介绍 和 设计目的
```

额外叮嘱：请了解 "拜占庭将军问题"， TCC

3. 本次课程内容预告

今天的主要内容有：

```
1、ZooKeeper的核心功能-ZNode数据模型
2、ZooKeeper的核心功能-Watcher
3、Zookeeper实战
    应用场景
    实际企业案例的实现
```

企业应用案例的目的：

```
1、自己会实现这些需求
2、要知道一些常用的流行的分布式技术底层使用zk到底干了什么
```

HBase， Spark 实现 HA 就是 基于 ZK 。 到时候看源码的时候，比较简单的介绍就可以了。！

4. 作业

作业：实现一个文件系统（类似于 zk 的数据模型）

验收要求：

```
1、实现基本的树形结构（DataTree），并且具备节点的增删改查等相关功能
    如果可以，帮我实现文件夹的剪切/移动
    DataTree dt = new DataTree()
    dt.insert(new DataNode())
2、具备冷启动数据恢复的功能
    具备把 内存中的数据  和 磁盘中的数据做同步和交换的 功能
    class ZKDatabase{
        private DataTree dt;
        private FileTxnSnapLog fileTxnSnapLog;
    }
    class FileTxnSnapLog{
        private TxnLog txnLog;       // 记录事务的操作日志的
        private Snapshot sh;         // 针对 DT 拍摄快照的
    }
    Namenode的元数据管理机制：
```

5. ZooKeeper核心功能

ZK 是一个分布式协调服务，劝架者，仲裁机构。 多个节点如果出现了意见的不一致，需要一个中间机构来调停！

ZK 就是一个小型的议会！当分布式系统中的多个节点，如果出现不一致，则把这个不一致的情况往 zk 中写。zk 会给你返回写成功的响应。但凡你接收到成功的响应，意味着 zk 帮你达成了一致！

但是事实上，如果你需要做 数据在多节点的状态同步，你把数据往 zk 写即可。写成功，意味着达成一致！

ZK 内部有一个数据系统！就是一棵树。树中的节点，就是一次事务的结果！

5.1. ZNode数据模型

FileSystem 类文件系统（每个节点要么是directory 要么是file）

ZNode系统/ZooKeeper数据模型（节点只有 znode（实现类：DataNode） 这一个名称，但是既有文件夹的能力，也有文件的能力）

两个方面：

1、znode的约束（znode 的节点存储的最大数据是1M，最好不要超过 1kb）为什么？

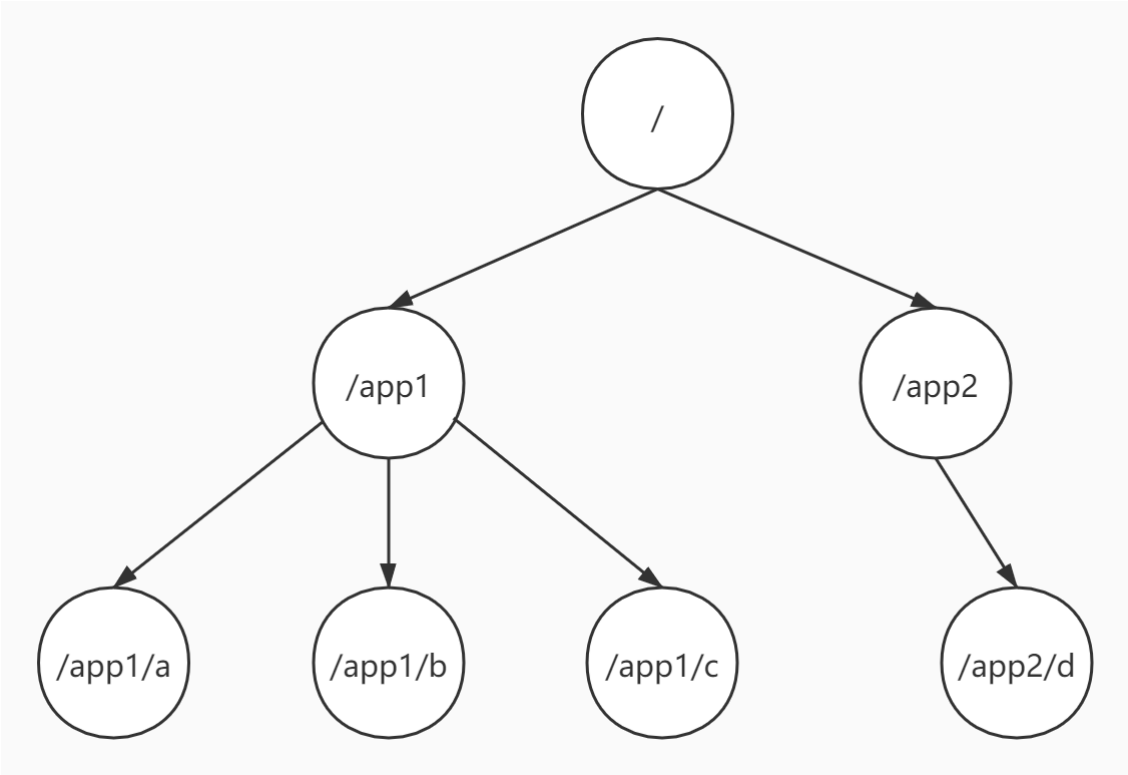
```
每个节点都有相同的数据视图：每个节点都存储了这个zk中的所有数据，每个节点的数据状态都和leader保持一致
1、同步的压力
2、存储的压力
```

2、znode的分类

```
1、按照生命周期
    持久类型（显示创建，显示删除，只有当使用显示命令来删除节点，否则这个节点知道被创建成功，则会一直存在）
    临时类型/短暂类型（跟会话绑定，那个会话创建的这个节点，如果这个会话断开，则这个会话创建的所有临时节点被系统删除）
2、按照是否带序列编号分，每个节点都各自维护了一个序列编号，当前节点的序列编号是由它的父节点维护的，编号是自增序列编号，和mysql的自增主键是一样的道理
    带
    不带
```

3、znode的小知识

```
临时节点的下面不能挂载子节点
```



如果自己要实现一棵树：

```
class DataNode{
    // 存储节点的数据
    private Object data;
    // 可以挂载一堆子节点
    private List<DataNode> children;
    // 当前节点有一个唯一的父节点，根节点除外
    private DataNode parent;
}
```

ZooKeeper 中，关于数据模型实现的几个重要 API

```
1、ZKDatabase          抽象出来的用于管理zk的整个完整的数据模型系统
    里头有两个重要的成员变量：
    1、FileTxnSnapLog          用于操作日志相关
        他也有两个重要的成员变量：
        1、TxnLog
        2、SnapShot
    2、DataTree                用于维护树形结构的数据模型
2、DataTree            树形结构的数据模型完整抽象，存在于内存中
3、DataNode            树中的节点
```

要思考为什么每个 znode 的存储最大是1M，推荐不要超过 1KB?

```
由于每个节点中存储的数据都是一样的，在进行写入操作成功的时候是需要所有节点同步成功
1、每个节点中存储的数据都是一样的： 如果所有数据的规模超出单台服务器的存储能力
2、写入过程至少要超过半数节点写成功才能认为该数据写成功
```

分类：

```
生命周期：临时节点EPHEMERAL      永久节点PERSISTENT
    默认创建永久节点，除非手动删除，否则一直存在
    每个znode节点必然都是由某一个session创建的。如果当前这个session断开，那么该znode节点会被自动删除
是否自带序列化SEQUENTIAL
    带序列化号
    不带序列化号
```

上面两种分类方式，两两组合，最终产生四种类型：

```
1、CreateMode.PERSISTENT
2、CreateMode.PERSISTENT_SEQUENTIAL
3、CreateMode.EPHEMERAL
4、CreateMode.EPHEMERAL_SEQUENTIAL
```

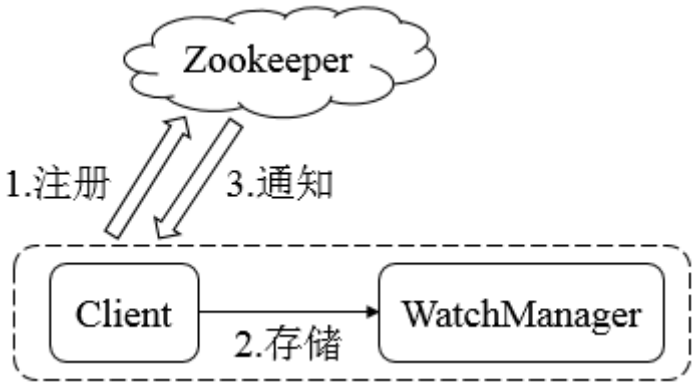
5.2. Watcher监听机制

ZooKeeper 的监听机制的工作原理：

```
1、ZooKeeper  链接对象  会话对象  客户端
    ZooKeeper zk = new ZooKeeper("bigdata02", 5000, null);
    zk.create()
    zk.delete()
    zk.setData()
    zk.getChildren() + zk.getData()

    注册监听：
    zk.getData(znodePath, watcher);    两件事：获取节点的数据 + 给当前节点注册了一个监听
    通俗的话理解：我执行这句话代码，也就意味着，我对zk系统中的 znodePath 节点的数据的变化感兴趣，所以我注册了一个监听（我告诉了 zk 系统，如果这个znode节点的数据发生了改变，我希望你能告诉我）。
    如果真的有其他的客户端更改了这个znode节点的数据，那么 zk 系统真的会通知我一个事件：WatchedEvent

2、WatchedEvent 事件通知对象
    KeeperState state  zk链接的状态
    String znodePath   发生事件的znode节点
    EventType type     事件的类型

3、当接收到这个 WatchedEvent 响应的时候，那么就应该响应的做一些逻辑处理， 这个逻辑处理的代码就写在回调方法中

4、客户端中，还存在一个 WatchManager 管理服务： 当前这个客户端到底注册了那些节点的那些监听都被分门别类的进行了管理。当这个客户端接收到 zk 系统的事件通知：WatchedEvent， 那么 WatchManager 就会根据这个事件对象内部的 znodePath + type + state 来确定后续操作是什么
```

监听对象：

```
interface Watcher{

    // 回调方法
    void process(WatchedEvent envet);
}
```



在应用方面几个重要的知识：

```
1、注册监听的方式
2、触发监听/触发事件的方式
3、事件的类型
```

1、注册监听的方式， 三种方式：

```
1、zk.getData()             关注节点的数据变化
2、zk.exists()              关注节点的存在与否的状态变化
3、zk.getChildren()         关注节点的子节点个数变化
```

2、触发监听的方式：

```
1、zk.setData               更改节点数据，触发监听
2、zk.create()              创建节点
3、zk.delete()              删除节点
```

3、响应的事件类型有四种：

```
1、NodeCreated                  节点被创建
2、NodeDeleted                  节点被删除
3、NodeDataChanged              节点数据发生改变
4、NodeChildrenChanged          节点的子节点个数发生改变
```

汇总一下，就是如下这张图：

创建Watch的API	触发事件				
	Create		Delete		setData
	znode	child	znode	child	znode
exists()	NodeCreated		NodeDeleted		NodeDataChanged
getData()			NodeDeleted		NodeDataChanged
getChildren()		NodeChildrenChanged	NodeDeleted	NodeChildrenChanged	NodeDataChanged

NodeCreated	创建节点时触发该事件
NodeChildrenChanged	当前节点下创建或者删除子节点触发该事件，修改子节点的数据不触发该事件
NodeDeleted	节点删除时触发该事件
NodeDataChanged	当前节点的数据被修改的时候触发该事件

最后一个小问题：请问，怎么实现循环监听/连续监听？因为 ZooKeeper 的监听只会响应一次

5.3. Session会话机制

ZK 系统中，依然有 session 的概念！

每次创建一个 DataNode(ZNode) 的时候，znode对象中， 有一个属性： State

State 中也有一个属性：owner 所有者， 这个znode节点是谁创建的，那么这个owner就是存储的谁！

谁：会话（如果节点是临时节点，如果是持久节点，这个 znode 的 state 属性的 owner 属性就是：0）

当把 owner 存储为 sessionID 的时候，如果这个 session 断开了，那么这个 sessionID 对应的所有 znode 都会被 zk 系统删除

当你创建临时节点的时候，会把创建这个节点的会话的ID保存在 znode 对象中的 state 属性中的 owner 中。

ZooKeeper 实例： 链接，会话，客户端（方便大家理解）

```
zookeeper在被实例化的时候，会创建两个对象：
1、网络通信的客户端对象（存在于客户端：ClientCnxn）
2、会话对象（存在于服务端）
```

ZK 的网络通信系统：

```
1、网络通信的服务端通信组件实现：ServerCnxn
2、网络通信的客户端通信组件实现：ClientCnxn
```

5.4. ZK编程模型

关于 zk 编程的一些细节知识：

```
1、先实例化一个客户端
    Zookeeper zk = new Zookeeper("bigdata02:2181,bigdata03:2181", 5000, null){
        ClientCnxn client = new ClientCnxn()
        client.send(ConnectRequest)  // 发送链接请求给 zk 系统
```

```
    }

2、通过 zk 实例，来进行各种操作
    zk.create()
    zk.delete()
    zk.getData()
    zk.exists()
    zk.getChildren()

3、一台服务器：QuorumPeer
    启动过程中，会创建  QuorumPeer 实例，在这个实例的内部会创建： ServerCnxn
    ServerCnxn 接收到 ConnectRequest ，就会执行链接处理，然后创建 Session

4、通信组件的实现：
    服务端：ServerCnxn
    客户端：ClientCnxn

5、关于初始化 Zookeeper 实例的时候，到底是链接的那个服务器呢？
    1、这句代码的内部会解析第一个参数：bigdata02:2181,bigdata03:2181 得到多台服务器列表
    随机取，具体规则是：通过 Collectoins.shuffle(adresses)[0]
    2、每个 Zookeeper 这个实例的创建，都只会链接一个节点，如果这个节点链接不上，如何重试呢?
    当 Zookeeper 和上一个节点断开了链接，则这个 客户端会尝试和另外一个节点建立连接（）
```

关于API的抽象的问题

```
1、HDFS
    Configuratioin + FileSystem
2、MapReduce
    InputFormat + RecordReader + Mapper + Partitioiner + Combiner + Reducer +
OutputFormat + RecoredWriter + Writable + Comparable + Comparator
3、ZooKeeper
    ZooKeeper + Watcher
4、Spark 非常的棒(编程入口， 数据抽象，算子)
    sparkContext sqlContext streamingContext
    rdd, dataFrame, dataset
    map, filter
5、HBase作为一个非常复杂的数据系统，对于对应的概念的抽象也做的不错
    1、配置对象：Coniguratioin
    2、链接对象：Connection
    3、管理对象：Admin（DDL） HTable（DML）
    4、操作对象：Put Delete Get Scan
    5、结果对象：ResultSet Result KeyValue/Cell
    ...
```

6. ZooKeeper 实战

6.1. ZooKeeper的应用场景

ZK： 分布式协调服务，劝架者，仲裁机构。基于它提供的两大核心功能：可以实现分布式场景中的各种疑难杂症！最经典的分布式锁的问题。只要基于 zk 很容易做实现！

```
1、发布/订阅 = 即时感知
2、命名服务
3、配置管理
4、集群管理
5、分布式锁
6、队列管理
7、负载均衡
```

不要一随便碰到一些疑难问题，就找zk！不要这么干！

经典的用法：尽量少的往 zk 中写数据，写入的数据也不要特别的大！只适合用来存储少量的关键数据！一个集群中到底谁是真正 active leader

```
1、因为每个节点都会做同步，在执行写请求的时候，事实上就是原子广播。
2、所有的请求，都是严格的顺序串行执行，这个zk集群在某一个时刻只能执行一个事务
3、zk 内部有一个leader，有少量的follower，可能有部分 observer，当 observer和 follower
接收到写请求，则会转发给 leader 来处理， 读请求，每个节点都可以处理。
所以当有大量的读请求的时候，只要扩展 observer 即可！
```

6.1.1. 发布订阅

举例说明：

```
1、系统          腾讯新闻App（提供很多的新闻频道）100个频道 新闻分类
2、发布者          小编（生产者）
3、订阅者          读者（消费者）
```

应用服务器集群可能存在两个问题： 1、因为集群中有很多机器，当某个通用的配置发生变化后，怎么自动让所有服务器的配置同一生效？ 2、当集群中某个节点宕机，如何让集群中的其他节点知道？ 为了解决这两个问题，zk引入了watcher机制来实现发布/订阅功能，能够让多个订阅者同时监听某一个主题对象，当这个主题对象自身状态发生变化时，会通知所有订阅者。

数据发布/订阅即所谓的配置中心：发布者将数据发布到zk的一个或一些列节点上，订阅者进行数据订阅，可以即时得到数据的变化通知。

发布/订阅有2种设计模式，推Push & 拉Pull。在推模中，服务端将所有数据更新发给订阅的客户端，而拉是由客户端主动发起请求获取最新数据。通常采用轮寻。

zk采用推拉结合，客户端向服务端注册自己需要关注的节点，一旦该节点数据发生变更，服务器像客户端发送Watcher事件通知，收到消息主动向服务端获取最新数据。**这种模式主要用于配置信息获取同步。**

A有一条消息，要让B知道：A主动告诉B， B不断的来询问有没有新的消息：如果有A就会告诉他

6.1.2. 命名服务

zookeeper 系统中的每个 znode 都有一个绝对唯一的路径！所以只要你创建成功了一个znode节点，也就意味着，你命名了一个全局唯一的 名称！

命名服务是分布式系统中较为常见的一类场景，分布式系统中，被命名的实体通常可以是集群中的机器、提供的服务地址或远程对象等，通过命名服务，客户端可以根据指定名字来获取资源的实体、服务地址和提供者的信息。ZooKeeper也可帮助应用系统通过资源引用的方式来实现对资源的定位和使用，广义上的命名服务的资源定位都不是真正意义上的实体资源，在分布式环境中，上层应用仅仅需要一个全局唯一的名字。**ZooKeeper可以实现一套分布式全局唯一ID的分配机制。**

由于zk可以创建顺序节点，保证了同一节点下子节点是唯一的，所以直接按照存放文件的方法，设置节点，比如一个路径下不可能存在两个相同的文件名，这种定义创建节点，就是全局唯一ID

专门用来做命名服务的算法：SnowFlake 雪花算法

6.1.3. 集群管理

HDFS中的datanode如果死掉了，那么naemndoe需要经过至少630s的默认时间，才会认为这个几点死掉！

所谓集群管理无在乎两点：**是否有机器退出和加入、选举master。**

对于第一点，所有机器约定在父目录 GroupMembers 下创建临时目录节点，然后监听父目录节点的子节点变化消息。一旦有机器挂掉，该机器与 ZooKeeper 的连接断开，其所创建的代表该节点存活状态的临时目录节点被删除，所有其他机器都将收到通知：某个兄弟目录被删除，于是，所有人都知道：有兄弟节点挂掉了。新机器加入也是类似，所有机器收到通知：新兄弟目录加入，又多了个新兄弟节点。

对于第二点，我们稍微改变一下，所有机器创建临时顺序编号目录节点，每次选取编号最小的机器作为 master 就好。当然，这只是其中的一种策略而已，选举策略完全可以由管理员自己制定。在分布式环境中，相同的业务应用分布在不同的机器上，有些业务逻辑（例如一些耗时的计算，网络I/O处理），往往只需要让整个集群中的某一台机器进行执行，其余机器可以共享这个结果，这样可以大大减少重复劳动，提高性能。

利用ZooKeeper的强一致性，能够保证在分布式高并发情况下节点创建的全局唯一性，即：同时有多个客户端请求创建 /currentMaster 节点，最终一定只有一个客户端请求能够创建成功。利用这个特性，就能很轻易的在分布式环境中进行集群选取了。（其实只要实现数据唯一性就可以做到选举，关系型数据库也可以，但是性能不好，设计也复杂）

6.1.4. 分布式锁

锁：并发编程。保证线程安全（一个JVM内部的多条业务逻辑的并发执行的安全）！针对临界资源直接进行加锁的操作。 谁来操作，都需要先拿到钥匙！拿到操作许可！这个操作许可同时只能一个人拿到。

一个分布式系统中的多个节点的并发执行的安全

有了 ZooKeeper 的一致性文件系统，锁的问题变得容易。

锁服务可以分为两三类： 独占锁，共享锁，时序锁

```
1、写锁：对写加锁，保持独占，或者叫做排它锁，独占锁
2、读锁：对读加锁，可共享访问，释放锁之后才可进行事务操作，也叫共享锁
3、时序锁：控制时序
```

对于第一类，我们将 ZooKeeper 上的一个 znode 看作是一把锁，通过 createznode() 的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。用完删除掉自己创建的 /distribute_lock 节点就释放出锁。

对于第二类，/distribute_lock 已经预先存在，所有客户端在它下面创建临时顺序编号目录节点，和选 Master 一样，编号最小的获得锁，用完删除，依次有序

6.1.5. 队列管理

两种类型的队列：

```
1、同步队列/分布式屏障：当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达。
2、先进先出队列：队列按照 FIFO 方式进行入队和出队操作。
```

第一类，在约定目录下创建临时目录节点，监听节点数目是否是我们要求的数目。
第二类，和分布式锁服务中的控制时序场景基本原理一致，入列有编号，出列按编号。

6.1.6. 负载均衡

ZooKeeper 实现负载均衡本质上是利用zk的配置管理功能，实现负载均衡的步骤：

```
1、服务提供者把自己的域名及IP端口映射注册到zk中。
2、服务消费者通过域名从zk中获取到对应的IP及端口，这里的IP及端口可能有多个，只是获取其中一个。
3、当服务提供者宕机时，对应的域名与IP的对应就会减少一个映射。
4、阿里的 dubbo 服务框架就是基于zk实现服务路由和负载。
```

6.2. Zookeeper最佳企业应用

6.2.1. 选举

见代码，包含实现思路！

```
/cluster_ha
    /active
        /hadoop01
    /lock
    /standby
        /hadoop02
        /hadoop03
```

```
/cluster_ha
    /active
        /hadoop02
    /lock
    /standby
        /hadoop03
```

思路实现：

```
前期：zk的状态：有/active节点和/standby节点，但是下面都没有信息的，/lock节点是不存在的
1、hadoop01一上线，发现自动成为standby角色，所以把自己的信息注册到 /standby 节点下
2、hadoop01一上线，再去找/active节点下是否有子节点，如果有就证明有active的角色，如果没有，去争抢分布式锁
3、如果没有抢到，意味着，别人抢到了，别人成为 active，如果自己抢到了，则把自己的信息在 /standby 里面删掉，再更新到 /active 节点下面
4、hadoop02一上线，发现/active 节点下面有子节点，有active，就自动成为 standby 角色，会监听 /active znode 监听：NodeChildrenChagended. 相当于告诉 zk 系统，只要 /active 的子节点个数个数发生变化，系统就告诉我一声。 /active 下面只能有一个子节点。既然减少，那么hadoop02就会收到通知
5、hadoop03上线，行为和 hadoop02 一致的
6、假设hadoop01宕机，hadoop01跟zk系统维持的会话就断开了额，由于创建的锁和 /active 下面的子节点都是临时节点，当hadoop01一宕机，这两个节点就自动被zk系统删除了， 由于hadoop02和hadoop03监听了这个事件，都会收到通知，则他们都知道了active leader不在了，他们都去争抢成为 active 先去抢 /lock 锁。 最终创建这个 lock 锁成功的只会是一台服务器，所以谁创建成功，谁就成为 active，没有创建锁节点 /lock 成功的就还是 standby.
```

推荐：使用 zookeeper的 API 框架： curator 来实现 HA： LeaderLatch LeaderSelector , 减轻代码编写的复杂度

6.2.2. 分布式锁

见代码，包含实现思路！

利用分布式锁可以实现：选举。 HDFS 的 HA 机制的实现，和我的 代码是一样的逻辑！

但是这并不是HDFS 实现 HA 的难点： 真正的难点： active 和 保持数据状态一致！

6.2.3. 配置管理

需求：一个集群中，有多个节点，如果更改一个参数，每个节点都得知道。

常用的方式：

```
1、写个shell脚本，通知所有节点
2、使用手动的方式，到每个节点更改
3、基于zk的监听来做： 更改配置的角色叫做客户端，注册监听感受参数变化的叫做：服务端
```

见代码，包含实现思路！

7. 总结

今天主要的内容，就是讲解，如何利用 Zookeeper 的两大核心功能，实现常见的一些企业需求。重点了解清楚 ZK 的工作原理，然后总结出套路：

- zookeeper 是对等架构，工作的时候，会举行选举，变成 leader + follower 架构
- zookeeper 中的所有数据，都在所有节点保存了一份完整的。
- zookeeper 的所有事务操作在zk系统内部都是严格有序串行执行的。
- zookeeper 系统中的leader角色可以进行读，写操作

- zookeeper 系统中的follower角色可以进行读操作执行，但是接收到写操作，会转发给leader去执行。
- zookeeper 系统的leader就相当于是一个全局唯一的分布式事务发起者，其他所有的follower是事务参与者，拥有投票权
- zookeeper 系统还有一种角色叫做 observer，这个角色和follower最大的区别就是 observer 除了没有选举权 和被选举权以外，其他的和 follower 完全一样
- observer 的作用是 分担整个集群的读数据压力，同时又不是增加分布式事务的执行压力，因为分布式事务的执行操作，只会在 leader 和 follower 中执行。observer 只是保持跟 leader 的同步，然后帮忙对外提供 读数据服务
- zookeeper 系统虽然提供了存储系统，但是这个存储，只是为自己实现某些功能做准备的，而不是提供出来，给用户存储大量数据的
- zookeeper 提供了znode节点的常规的增删改查操作，使用这些操作，可以模拟对应的业务操作，使用监听机制，可以让客户端立即感知这种变化。
- zookeeper集群和其他分布式集群最大的不同，在于zk是不能进行线性扩展的。因为像 HDFS 的集群服务能力是和集群的节点个数成正比，但是zk系统的节点个数越多，反而性能越差
- zookeeper 集群的最佳配置：比如9,11,13个这样的 follower节点，observer 若干！follower 切记不宜太多！
- zookeeper 系统如果产生了这么一种情况：某个znode的数据变化非常的快，每次变化触发一次process回调！由于zk执行事务的时候，是串行单节点严格有序执行的。leader负责这个事务的顺序执行。多个事件来不及执行，上一个事件还没有执行，下个触发动作，zk会忽略！影响不大！